Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 70 (2008) 221–240
Science of Computer Programming
www.elsevier.com/locate/scico

Mobile JikesRVM: A framework to support transparent Java thread migration

Raffaele Quitadamo*, Giacomo Cabri, Letizia Leonardi

Dipartimento di Ingegneria dell'Informazione, Università di Modena e Reggio Emilia, Via Vignolese, 905 – 41100 Modena, Italy

Received 1 December 2006; received in revised form 1 February 2007; accepted 11 July 2007
Available online 1 November 2007

Abstract

Today's complex applications must face the distribution of data and code among different network nodes. Computation in distributed contexts is demanding increasingly powerful languages and execution environments, able to provide programmers with appropriate abstractions and tools. Java is a wide-spread language that allows developers to build complex software, even distributed, but it cannot handle the migration of computations (i.e. threads), due to intrinsic limitations of many traditional JVMs. After analyzing the approaches in the literature, this paper presents our thread migration framework (called *Mobile JikesRVM*), implemented on top of the IBM Jikes Research Virtual Machine (RVM): exploiting some of the innovative techniques in the JikesRVM, we implemented an extension of its scheduler that allows applications to easily capture the state of a running thread and makes it possible to restore it elsewhere (i.e. on a different hardware architecture or operating system), but still with a version of the framework installed). Our thread serialization mechanism provides support for both proactive and reactive migration, available also for multi-threaded Java applications, and tools to deal with the problems of resource relocation management. With respect to previous approaches, we implemented Mobile JikesRVM without recompiling its JVM (Java Virtual Machine) source code, but simply extending JikesRVM functionalities with a full Java package to be imported when thread migration is needed.

Keywords: Java virtual machine; Thread migration; Code mobility; JikesRVM; Distributed applications

1. Introduction

Modern distributed systems [28] are becoming more and more complex. This has lead to the need for flexibility, that has to be considered very desirable, if not mandatory, when large scale distributed computations are performed. Conventional software components, scattered among network nodes, provide services to other components or to end users, but are often statically bound to their hosting environment. This view is being challenged by technical developments that introduce a degree of mobility in distributed systems. Wireless LANs and mobile devices have already highlighted the potentials of physical mobility [17,4]. *Code mobility* [12] is instead reshaping the logical structure of modern distributed systems as it enriches software components (in particular, execution units) with the

* Corresponding author.
E-mail addresses: quitadamo.raffaele@unimore.it (R. Quitadamo), cabri.giacomo@unimore.it (G. Cabri), leonardi.letizia@unimore.it (L. Leonardi).

0167-6423/$ - see front matter
doi:10.1016/j.scico.2007.07.009

capability to dynamically reconfigure their bindings with the underlying execution environments. The concept is simple and elegant; an object (that may be active or passive) that resides on one node is migrated to another node where execution is continued. The main advantages of mobile computations, whether they are object-based or not, are as follows:

1. *load balancing*: distributing computations among many processors as opposed to computations on one processor gives faster performance for tasks that can be fragmented;
2. *communication performance*: active entities that interact intensively can be moved to the same node to reduce the communication cost for the duration of their interaction;
3. *availability*: entities can be moved to different nodes to improve the service and provide better failure coverage or to mitigate against lost or broken connections;
4. *reconfiguration*: migrating entities permits continued service during upgrade or node failure;
5. *location independence*: an entity visiting a node can rebind to generic services without needing to specifically locate them.

It can be argued that Java has all the features to make thread mobility possible thanks to its platform-independent bytecode language and the support for object serialization. Regular Java objects can be easily made persistent or migrated to other machines, by means of the JVM built-in serialization facility. Java threads are coherently presented to the programmer as objects as well, but their serialization does not produce the desired effect of "capturing their current execution flow and resuming it elsewhere"; it is just the `java.lang.Thread` object, together with its fields, that is serialized, while the real execution flow is still tightly bound to the execution environment. The Java programming language does not therefore support the migration of threads, which exists for other languages and specific operating systems [19].

Some kind of framework or "JVM enhancement" is thus needed to enable mobile computations in distributed Java applications. Several approaches have been proposed and experimented with in order to add thread migration capability to the Java run-time environment. The bulk of them provides only a "weaker form" of mobility, where the system simply allows the migration of the code and some data, but discards the execution state (i.e. the Java method stack, context registers and instruction pointer). In Section 2 of this paper, we argue that *weak mobility* is not always the best choice (e.g. particularly when dealing with complex parallel computations) and is definitely unsuitable in some cases (e.g. if the program has a recursive behavior). Section 2 describes some important motivations for strong thread mobility, sketching some potential applications as well. After having shortly discussed the main issues of thread migration in the literature (in Section 3), in Section 4 we outline the main contributions of this paper:

1. a *novel approach towards the provision of Java thread strong migration*, integrated into a full-fledged framework, built on top of the IBM Jikes Research Virtual Machine. Such a framework, called *Mobile JikesRVM* [18], exploits some well-established object-oriented language techniques (e.g. On-Stack Replacement, type-accurate garbage collectors, quasi-preemptive Java thread scheduler, etc.) to capture the execution state of a running thread, in both proactive and reactive situations; it also provides the programmer with a dedicated resource management layer, capable of handling issues like object binding reconfigurations, inter-thread references and synchronization upon migrated threads;
2. *an extension of the IBM JikesRVM scheduler* that Java programmers can dynamically enable, simply importing a Java package into their mobile Java applications.

The choice of JikesRVM is strongly motivated by the fact that it was born as a VM specifically targeted to multiprocessor SMP servers [1]. Likewise, our framework focuses on these kinds of hardware, where strong mobility is mostly required. In Section 5, we present our performance tests, made writing a benchmark based on the Fibonacci recursive algorithm, and in Section 6 we prospect our future research work on this mobility framework. Conclusions are drawn in Section 7.

2. Background and motivations

This section introduces the main issues to address when designing a thread migration mechanism and sketches some real applications that would benefit from the work explained in this paper.

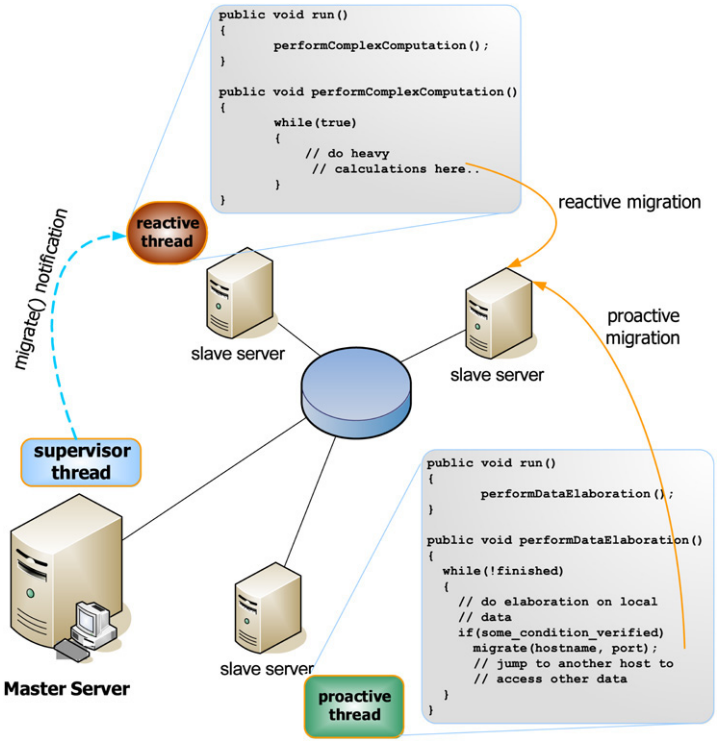

Fig. 1. A cluster of servers to perform distributed computations.

2.1. Motivation

The choice of thread mobility, when designing distributed Java applications, has to be carefully motivated, since it is not always the best one in most simple cases: e.g. many mobile agents applications [10] do not require such a big support for computations migration, relying on simpler form of data migration. The category of distributed and parallel computations can be considered perhaps the "killer application" for thread mobility. For instance, complex computations, possibly with a high degree of parallelism, carried out on a cluster of servers would certainly benefit from a thread migration facility in the JVM (see Fig. 1). Well-known cases of such applications are mathematical computations, which are often recursive by their own nature (e.g. fractal calculations) and can be parallelized to achieve better elaboration times.

In the cluster depicted in Fig. 1, several mobile threads are spawned by a supervisor thread on a master server and each one can be assigned a portion of a huge data space (e.g. temperature and pressure values from different geographic areas, in a weather forecast application). In order to cover the entire data domain, each mobile thread can exploit the migration support by the underlying JVM to move spontaneously (i.e. proactively in Fig. 1) from one server (initially the master) to another slave server, without having to restart from the beginning (i.e. *strong mobility* [10]). It simply carries its current call stack with itself and continues execution at destination from the last executed instruction.

In a similar scenario, we can have mobile threads moved reactively, i.e. after being notified a migration request from some other thread. This is the case of distributed systems (e.g. in the Grid Computing field) where *load balancing* is intensively carried out and reactive thread relocation is a must: in such case a number of slave nodes (like those in Fig. 1) have several tasks assigned to them. In order to avoid overloading some nodes while leaving some others idle (for a better exploitation of the available resources and an increased throughput), these systems need to constantly monitor the execution of their tasks and possibly re-assign them, according to an established load balancing algorithm. In the latter systems, it frequently happens that a supervisor thread manages the workload of each slave server, implementing some load balancing algorithm. The supervisor thread in Fig. 1 should be able to notify one spawned mobile thread to move on a less overloaded server: this clearly stands for preempting its execution at some point in the code, moving its captured state at the destination host and transparently resuming it. As we will see later, a particular

```
public class MyAgent extends Agent {

protected boolean migrated = false;
// indicates if the agent has moved yet

public void run(){
   if( ! migrated ){
      //…things to do before the migration
      migrated = true;
      try{ migrate(newURL("nexthost.unimore.it");
      }catch(Exception e){ migrated = false; }
   }
   else{ // things to do on the destination host
      // possibly other if/else to handle other
      // conditions…
      }
}}
```

Fig. 2. An example of a weak mobile agent.

kind of thread migration (called *reactive migration*), that we provide in our framework, fits very well the requirements of these systems.

These kinds of well-known parallel systems clearly pose challenging requirements on the underlying execution environments, because they need a framework capable of:

1. *capturing in a portable format the full execution state of a running thread* (i.e. code, objects but, more importantly, the frames in the call stack) and restoring it on another JVM. Portability is desirable if we want to address heterogeneity of platforms (OS) and architectures (i.e. Intel, PPC, . . .);
2. *allowing migration to be carried out both proactively and reactively*. This requires strong cooperation with the scheduler underneath;
3. *properly handling resource relocation and binding reconfiguration*. This implies giving the programmer some tools (e.g. the relocation policy presented later) to specify which resources/objects to bring with the thread, how he/she intends to manage the bindings with such resources after migration (e.g. by copy, by network reference and so on).

The discussion in the next subsection goes deeper into these problems, while Section 3 describes how they have been addressed in the literature, highlighting current approaches, and drawbacks that we tried to overcome.

2.2. Thread mobility issues

Java threads are often considered a valid example of so-called execution units [12], performing their tasks in a computational environment (i.e. the JVM), but without any possibility of detaching from their native environment. An execution unit is conventionally split into three separate parts that are supposed to be movable to achieve the overall mobility of the execution unit:

1. the *code segment* (i.e. the set of compiled methods of the application);
2. the *data space*, a collection of all the resources accessed by the execution unit. In an object-oriented system, these resources are represented by objects in the heap;
3. an *execution state*, containing private data as well as control information, such as the call stack and the instruction pointer.

Weakly mobile threads [10] can transfer their execution, bringing only code and some data, while the call stack is lost. From the architectural standpoint, it is relatively easy to implement weak mobility on top of the JVM, because the Java language provides very powerful tools for that purpose: object serialization is used to migrate data, such as objects referenced by the program; bytecode and dynamic class-loaders facilitate the task of moving the code across distant JVMs, hosted by heterogeneous hardware platforms and operating systems.

From an application point of view, weakly mobile systems usually force the programmer to write code in a less natural style: extra programming effort is required in order to manually save the execution state, with flags and other artificial expedients. For instance, Mobile Agents (MA) [10] are usually weakly mobile execution units, used in many scenarios: e.g. distributed information retrieval, online auctions and other systems where they have to follow the user's movements, for migrating from and to the user's portable device (mobile phone, PDA, etc.). A simple weak agent is shown in Fig. 2. The point is that, with weak mobility, it is as the code routinely performs rollbacks. In fact, looking at the code in Fig. 2, it is clear how, after a successful `migrate()` method call that causes the agent migration, the code

does not continue its execution in the `run()` method from that point. Instead, the code restarts from the beginning of the `run()` method (on the destination machine, of course), and thus there is a code rollback. The fact that an agent restarts its execution always from a defined entry point, could produce awkward solutions, forcing the developer to use flags and other indicators to take care of the host the agent is currently running on.

A strongly mobile thread has instead the ability to migrate its code and execution state, including the program counter, saved processor registers, return addresses and local variables. The active component is suspended, marshaled, transmitted, unmarshaled and then restarted at the destination node without loss of data or execution state. In a previous work on mobile agents [21], we experimented with the use of our thread migration framework, to endow agents (using the IBM Aglets mobile agent platform) with strong mobility support.

Strong mobility turns out to be far more powerful where complex distributed computations are required, as it preserves the traditional programming style of threads, without requiring any code rollback or other expedients: it reduces the migration programming effort to the invocation of a single operation (e.g. a `migrate()` method) and leads to cleaner implementations. Despite these advantages, many systems support only weak mobility and the reason lies mainly in the complexity issues of strong mobility and in the insufficient support of existing JVMs to deal with the execution state. Moreover, a weakly mobile system gives the programmer more control over the amount of state that has to be transferred, while an agent using strong migration may bring unnecessary state, increasing the size of the serialized data.

3. Related work

Several approaches have been proposed so far to overcome the limitations of the JVM as concerns the execution state management. The main decision that each approach has to take into account is how to capture the internal state of threads, providing a fair trade-off between performances and portability. In the literature, we can typically find two categories of approaches:

1. modifying or extending the source code of existing JVMs to introduce APIs for enabling migration (*JVM-level approach*);
2. translating somehow the application's source code in order to trace constantly the state of each thread and using the gathered information to rebuild the state remotely (*application-level approach*).

3.1. JVM-level approach

The former approach is, with no doubt, more intuitive because it provides the user with an advanced version of the JVM, which can completely externalize the state of Java threads (for thread serialization) and can, furthermore, initialize a thread with a particular state (for thread de-serialization). The kind of manipulations made upon the JVM can be several.

The first proposed projects following the JVM-level approach like Sumatra [26], Merpati [25], JavaThread [6,7] and NOMADS [27], extend the Java interpreter to precisely monitor the execution state evolution. They, usually, face the problem of stack references collection modifying the interpreter in such a way that each time a bytecode instruction pushes a value on the stack, the type of this value is determined and stored "somewhere" (e.g., in a parallel stack). The drawback of this solution is that it introduces a significant performance overhead on thread execution, since additional computation has to be performed in parallel with bytecode interpretation. Other projects tried to reduce this penalty avoiding interpreter extension, but rather using JIT (Just In Time) re-compilation (such as Jessica2 [32]) or performing type inference only at serialization time (and not during thread normal execution). In ITS [5], the bytecode of each method in the call stack is analyzed with one pass at serialization time: the type of stacked data is retrieved and used to build a portable data structure representing the state. The main drawback of every JVM-level solution is that they implement special modified JVM versions that users have often to download; therefore they are forced to run their applications on a prototypal and possibly unreliable JVM.

3.2. Application-level approach

In order to address the issue of non-portability on multiple Java environments, some projects propose a solution at the application level. In these approaches, the application code is filtered by a pre-processor, prior to execution, and

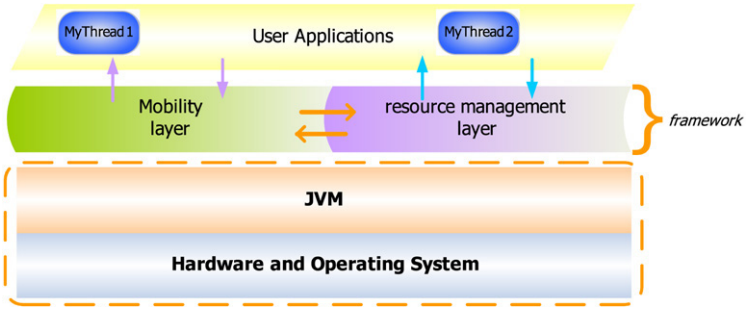

Fig. 3. A layered view of Mobile JikesRVM.

new statements are inserted, with the purpose of managing state capturing and restoration. Some of these solutions rely on a bytecode pre-processor (e.g. JavaGoX [22] or Brakes [30]), while others provide source code translation (e.g. Wasp [13], JavaGo [23], Wang's proposal [31]). Two of them [23,31] hide a weak mobility system behind the appearance of a strong mobility one: they, in fact, re-organize "strongly mobile" written code into a "weakly mobile" style, so that weak mobility can be used instead. Portability is achieved at the price of a slowdown, due to the many added statements.

3.3. Discussion

Starting from the above considerations, we have decided to design and implement a strong thread migration system able to overcome many of the problems of the above-explained approaches. In particular, our framework is written entirely in Java and it does neither suffer performance overheads, due to bytecode instrumentations, nor reliability problems, because the user does not have to download a new, possibly untrustworthy, version of JikesRVM. The framework is capable of dynamically installing itself on several recent versions of JikesRVM (we carried out successful tests starting from release 2.3.2). In fact, every single component of the migration system has been designed and developed to be used as a normal Java library, without requiring rebuilding or changing the VM source code. Therefore, our JikesRVM-based approach can be classified as a midway approach between the above-mentioned JVM-level and Application-level approaches. Other midway approaches [16] exploit the JPDA (Java Platform Debugger Architecture) that allows debuggers to access and modify run-time information of running Java applications. The JPDA (currently replaced by JVMTI [14]) can be used to capture and restore the state of a running program, obtaining a transparent migration of mobile agents in Java, although it suffers from some performance degradation due to the debugger intrusion.

4. Enabling thread migration on top of JikesRVM

In order to fulfill the requirements of many distributed and parallel applications, an efficient and well-designed "software support" is needed on top of the bare JVM. Such a middleware should provide a precise, but flexible and customizable, answer to the questions of parallel applications developers. In the next subsections, we follow a top-down approach towards the description of *Mobile JikesRVM* architecture.

4.1. A layered view of mobile JikesRVM

From a mere technological standpoint, the capability to move code and regular objects is already a consolidated matter, thanks to bytecode and dynamic class-loaders (which facilitate the task of moving the code across distant JVMs, hosted by heterogeneous hardware platforms and operating systems) and object serialization (used to migrate data in the heap). The main problem to tackle here is how to *detach the execution state of a Java thread from its native environment and then to re-install it at some other site*. This requires diving into the internals of the JVM core and externalizing a complete representation of the running thread. Such functionality is provided in our framework by the mobility layer in Fig. 3, which is built just upon the JVM. This layer should be hopefully pluggable dynamically into the JVM run-time, without requiring heavy modifications of the JVM source code, which would probably affect the

performance of non-mobile threads as well. Further details on this layer and its interactions with the JVM are the subject of the next subsection.

Shifting to a more application-level point of view, every mobility system (both weak and strong) will sooner or later run across the non-negligible issue of data space management [12]: every thread has a set of referenced objects into the heap (i.e. the data space) and, when it migrates to the destination site, the set of bindings to passive (i.e. resources) and active objects (i.e. other threads) has to be rearranged. The way this set is rearranged depends on the nature of the resources (for instance, whether they can be migrated or not over the network), the type of the binding to such resources, as well as requirements posed by the application. The very fact that it eventually depends on application specific requirements makes it impossible to fully automate the choice of the adequate strategy, entailing the need for its programmatic specification. The resource management layer in Fig. 3 is responsible for handling references to resources and relocating them according to such programmatic specifications. This layer is the subject of Section 4.3.

On top of the framework, it is possible to develop different distributed applications using the strong mobility and data space management support described in the next two subsections.

4.2. The mobility layer

In this section we will describe how we implemented our strong migration mechanism on top of the IBM Jikes Research Virtual Machine (RVM). The JikesRVM project was born in 1997 at the IBM T.J. Watson Laboratories [1] and it has been recently donated by IBM to the open-source community. Two main design goals drove the development of such successful research project [2]:

1. supporting high performance Java servers;
2. providing a flexible research platform "where novel VM ideas can be explored, tested and evaluated".

In this research virtual machine, several modern programming language techniques have been experimented with and, throughout this presentation, we will focus mainly on those features that are most strategic to our system. The proposed description will follow the migration process in its fundamental steps, from thread state capturing to resuming at the destination machine.

4.2.1. Our JikesRVM extension

When the programmer wants to endow her threads with the capability to migrate or be made persistent on disk, the first simple thing to do is to import the `mobility` package, which exposes the `MobileThread` class. The latter inherits directly from the `java.lang.Thread` class and has to be subclassed by user-defined threads.

The configuration of the scheduler, when our JikesRVM extension is installed, is reported in Fig. 4: the idea is that now, with mobility, threads can enter and exit the local environment through some migration channels. These channels are represented in Fig. 4 using the classical notation of queuing networks. The software components added by our infrastructure are highlighted with boxes and it can be clearly seen how these parts are dynamically integrated into JikesRVM scheduler, when the programmer enables the migration services.

The single output channel is managed by a regular service thread, which runs in a loop, performing the following actions:

1. extract a mobile thread from the migration queue, where these threads wait to be transferred;
2. establish a TCP socket connection with the specified destination (see the code in Fig. 5);
3. if the previous step is successful, then capture the complete thread state data (i.e. the `MobileThread` object and the sequence of frames into its call stack);
4. serialize those data into the socket stream (`java.io.ObjectOutputStream`);
5. close the connection with the other end-point.

One or more input channels are implemented by means of regular service threads listening on specific TCP ports. When a connection request is issued from the network, they simply do the following (see the code from Fig. 6):

1. open the connection with the requesting host;
2. read thread state data from the socket stream (`java.io.ObjectInputStream`);
3. re-establish a local instance of the arrived thread
4. install all the frames into the newly allocated stack

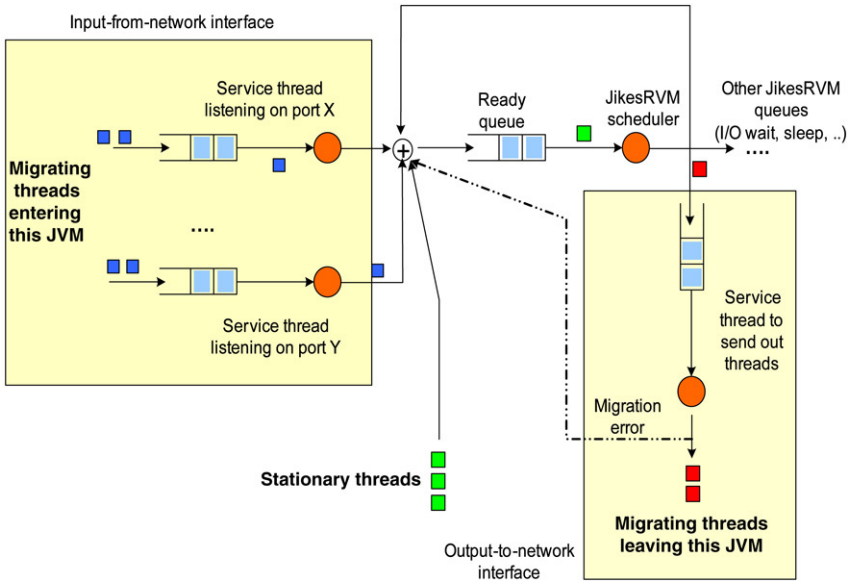

Fig. 4. Adding migration channels to JikesRVM scheduler.

```
private void migrate(String hostname, int port) {
   try {
       Socket sock = new Socket(hostname, port);

       TransportObject t = new TransportObject();
       t.thread=this;
       t.framesChain=collectFrames(...);

      ObjectOutputStream oos=new ObjectOutputStream(sock.getOutputStream());
       oos.writeObject(t);
       oos.flush();
       sock.close();} catch (Exception e) {...}
}
```

Fig. 5. An excerpt of the migrate() method from the output channel thread.

5. resume the recreated thread and close the connection socket.

A first observation is that non-mobile ("stationary" as in Fig. 4) threads in the system are not influenced at all by the infrastructure built upon the scheduler. Only those threads that inherit from our enhanced `MobileThread` class are interested in the added migration services.

4.2.2. Proactive migration vs. reactive migration

There are two ways for a `MobileThread` to get queued into the migration queue, waiting for the output channel to transfer it [12]:

1. the mobile thread autonomously determines the time and destination for its migration, calling the `MobileThread.migrate(URL destination)` method (*proactive migration*);
2. its movement is triggered by a different thread that can have some kind of relationship with the thread to be migrated, e.g. acting as a manager of roaming threads (*reactive migration*).

Exploiting JikesRVM features, we successfully implemented both migration types, in particular the reactive migration. As anticipated in Section 2.1, an application, in which reactive migration can be essential, is a load balancing facility in a distributed system. If the virtual machine provides such functionality to authorized threads, a load monitor thread may want to suspend the execution of a worker thread A, assign it to the least overloaded machine and resume its execution from the next instruction in A's code. This form of transparent externally requested migration is harder to implement with respect to the proactive case, mainly because of its asynchronous nature.

```
void handleTransportObject(ObjectInputStream ois, ObjectOutputStream oos){
       /* Read the object from the socket */
       TransportObject t = (TransportObject) ois.readObject();

       /* Get the received thread */
       MobileThread newThread = t.thread;

       /* Make this thread autosuspended... */
       newThread.enableMobileThread(true);
       newThread.start();

       /*...and wait for its suspension */
       while(!newThread.isAutoSuspended())
          Thread.yield();

       /* Install frames into the new thread */
       newThread.installFrames(t.framesChain);

       /* Resume the thread locally */
       newThread.resume();
}
```

Fig. 6. An excerpt from the code of the input channel thread.

Proactive migration raises, in fact, less semantic issues than the reactive one, though identical to the latter from the technological/implementation point of view: in both cases we have to walk back the call stack of the thread, extract the meaningful frames and send the entire thread data to destination (see the following two subsections for more details). The fundamental difference is that proactive migration is synchronized by its own nature (the thread invokes `migrate()` when it wants to migrate), while for reactive migration the time when the thread has to be interrupted could be unpredictable (the requester thread notifies the migration request to the destination thread, but the operation is not supposed to be instantaneous). Therefore, in the latter case, the critical design-level decision is about the degree of asynchronism to provide. In a few words, the question is: should the designated thread be interruptible anywhere in its code or just in specific safe migration points?

We chose to provide a more coarse-grained migration in the reactive case. Our choice has a twofold motivation:

1. designing the migration facility is simpler;
2. decreasing migration granularity reduces inconsistency risks.

Although these motivations can be considered general rules-of-thumb, they are indeed related to the VM we adopted. In fact, the scheduling of the threads in JikesRVM has been defined as quasi-preemptive [1], since it is driven by JikesRVM compilers. In JikesRVM, Java threads are objects that can be executed and scheduled by several kernel-level threads, called virtual processors, each one running on a physical processor. What happens is that the compiler introduces, within each compiled method body, special code (yieldpoints) that causes the thread to request its virtual processor if it can continue the execution or not. If the virtual processor grants the execution, the virtual thread continues until a new yieldpoint is reached, otherwise it suspends itself so that the virtual processor can execute another virtual thread. In particular, when the thread reaches a certain yieldpoint and the virtual processor informs it that its time slice has expired, the thread prepares itself to dismiss the scheduler and lets a context switch occur. The invoked function to deal with a reached yieldpoint is the static method `VM_Thread.yieldpoint()`. If we allow a reactive migration with too fine a granularity (i.e. potentially at any yieldpoint in thread's life), inconsistency problems are guaranteed: the latter is because the thread can potentially lose control in any method, from its own user-implemented methods to internal Java library methods (e.g. `System.out.println()`, `Object.wait()` and so forth). It may occur that a critical I/O operation is being carried out and a blind thread migration would result in possible inconsistency errors.

We are currently tackling the reactive migration issues thanks to JikesRVM yieldpoints and the JIT compiler. In order to make mobile threads interruptible with the mentioned coarse granularity, we introduced the migration point concept: migration points are always a subset of yieldpoints, because they are reached only if a yieldpoint is taken. The only difference is that migration points are inserted only:

1. in the methods of the `MobileThread` class (*by default*);
2. in all user-defined class implementing the special `Dispatchable` interface (*class-level granularity*);
3. in those user-methods that are declared to throw `DispatchablePragmaException` (*method-level granularity*).

```
static void migrationPoint() throws NoInlinePragma {

   if (migrationRequested){

   // Process a migration point
   outputChannel.queue(Thread.getCurrentThread());

   suspend(); // thread-switch beneficial

   // The thread is resumed here at the
   // destination
   }
   else // Process a regular  yieldpoint
   {
      yieldpoint();
   }
}
```

Fig. 7. The method that deals with a migration point.

The introduction of a migration point forces the thread to check also for a possibly pending migration request (notified reactively by another thread). If the mobile thread takes the migration point, it suspends its execution locally and waits in the migration queue (described in the previous paragraph) until the service thread, responsible for the output channel, selects it and starts the necessary migration operations (see the next two paragraphs). The code for these additional tests is partly reported in Fig. 7.

This approach has several advantages: firstly, it rids us of the problem of unpredictable interruptions in internal Java library methods (not affected by migration points at all); then, it also gives the programmer more control over the migration, by letting her select those safely interruptible methods; last but not least, it leaves the stack of the suspended thread in a well-defined state, making the state capturing phase simpler. We achieved the insertion of migration points, simply patching at run-time a method of the JIT compiler (the source code of the VM is left untouched and one can use every OSR-enabled version of the JikesRVM). As we already mentioned, yieldpoints are inserted by the JIT compiler, when it compiles a method for the first time. These yieldpoints are installed into method prologues, epilogues and loop heads by the `genThreadSwitchTest()` method of the compiler (in `com.ibm.JikesRVM.VM_Compiler`). In order to force the compiler to insert our migration points instead of yieldpoints in the methods listed above, we patched the `genThreadSwitchTest()` method with an internal method of the framework: the new method has a code nearly identical to the old one, except for the special treatment of the three cases listed above. In these cases, the thread enters the code in Fig. 7 and is auto-suspended, waiting to be serialized by the output channel described in the previous paragraph.

We must point out that JikesRVM's compiler does not allow unauthorized user's code to access and patch internal run-time structures. User's code, compiled with a standard JDK implementation, will not have any visibility of such low-level JikesRVM-specific details.

4.2.3. Capturing the execution state of a thread

When the service thread, owner of the output channel described earlier, selects a `MobileThread` candidate to serialize, it starts a walk back through its call stack, from the last frame to the `run()` method of the thread. This jump is shown schematically in Fig. 8, where the stack is logically partitioned into three areas:

1. *internal preamble frames*, which are always present and do not need to be migrated;
2. *user-pushed frames*, to be fully captured as explained in the next subsection;
3. *thread-switch internal frames*, which can be safely replaced at the destination and, thus, not captured at all.

A special utility class, called `FrameExtractor`, has been implemented in our framework, with the precise goal of capturing all the frames in the user area in a portable bytecode-level form. One interesting method of this class is the `extractSingleFrame()` method reported in the code of Fig. 9. This method uses an "OSR extractor" to capture the frame state representation and returns it to the caller, ready to be serialized and sent to destination or to be checkpointed on disk. The OSR extractor is described in the following paragraphs.

4.2.4. The JikesRVM OSR extractor

The OSR (On-Stack Replacement) extractor is another fundamental component of the framework: it takes inspiration from the OSR extractors provided by JikesRVM [11], though it has been re-written for the purposes of our project.

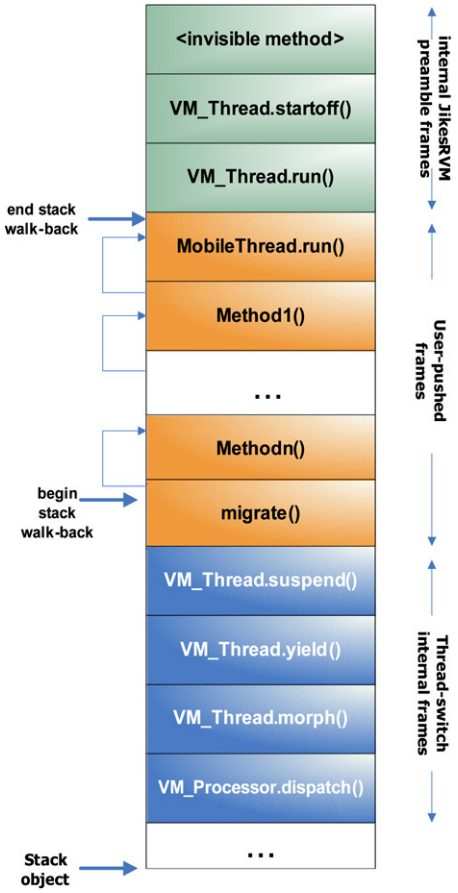

Fig. 8. The stack walk-back of a suspended `MobileThread`.

```
MobileFrame extractSingleFrame() {

  /* Move to the previous user frame (walk-back) */
   if(!moveToPreviousFrame())
    return null; //no more user frames to capture

  /* Extract the state in the specified object */
  return defOSRExtractor.extractState(...);

}
```

Fig. 9. The `extractSingleFrame()` method of the `FrameExtractor` class.

The OSR technique was introduced in JikesRVM, with a completely different objective from ours: enabling adaptive re-compilation of hot methods. In fact, JikesRVM can rely not only on a baseline compiler but also on an optimized one [8]. Every bytecode method is initially compiled with the baseline compiler, but when the Adaptive Optimization System (AOS) [3] decides that the current executing method is worth being optimized, the thread is drawn from the ready queue and the previous less-optimized frame is replaced by a new more-optimized frame. The thread is then rescheduled and continues its execution in that method. This technique was first pioneered by the Self programming language [9]. An innovative implementation of the OSR was integrated into the JikesRVM [11], which uses source code specialization to set up the new stack frame and continue execution at the desired program counter. The transition between different kinds of frames required the definition of the so-called JVM scope descriptor that is "the compiler-independent state of a running activation of a method" based on the stack model of the JVM [15]. When an OSR is triggered by JikesRVM, the scope descriptor for the current method is retrieved and is used to construct a method, in bytecode, that sets up the new stack frame and continues execution, preserving semantics.

```
class MobileFrame {

    /** Name of the method which adds this frame*/
    public String methodName;

    /** Method descriptor e.g. "(I)V" for a method getting an integer and
    returning void */
    public String methodDescriptor;

    /** Fully qualified method class(e.g."mypackage.myClass")*/
    public String methodClass;

    /** The bytecode index (i.e. return address) within this method*/
    public int bcIndex;

    /** The local bytecode-level local variable including parameters */
    public MobileFrameElement[] locals;

    /** The value of the stack operands at the specified bytecode index */
    public MobileFrameElement[] stack_operands;

    // methods and static fields omitted…
}
```

Fig. 10. The main fields of the `MobileFrame` class.

4.2.5. Our modified OSR extractor

The JikesRVM OSR frame extractor has been rewritten for the purpose of our mobility framework (we called it `OSR_MobilityExtractor`) to produce a frame representation, suitable for a thread migration context. The scenario we are talking about is a wide-opened one, where different machines running JikesRVM mutually exchange their `MobileThreads` without sharing the main memory. We introduced, therefore, a portable version of the scope descriptor, called `MobileFrame`, whose structure is reported in Fig. 10. While the OSR implementation in JikesRVM uses an internal object of class `VM_NormalMethod` to identify the method of the frame, we cannot make such an assumption; the only way to identify that method is through the triplet

⟨*method name, method descriptor, method class*⟩

that is supposed to be universally valid (the class should be a fully qualified class name, with a unique package name). This triplet (represented by the three fields `methodName`, `methodDescriptor` and `methodClass` in Fig. 10) is used to refer the method at the destination (e.g. its bytecode must be downloaded if not locally available yet), maybe after a local compilation.

The bytecode index (i.e. the `bcIndex` field) is the most portable form to represent the return address of each method body and it is already provided in JikesRVM by default OSR. Finally, we have two arrays (i.e. the `locals` and `stack_operands` fields) that, respectively, contain the values of local variables (including parameters) and stack operands in that frame. These values are extracted from the physical frame at the specified bytecode index and converted into their corresponding Java types (`int`, `float`, `Object` references and so on). In addition, it must be pointed out that the `OSR_MobilityExtractor` class fixes up some problems that we run across during our implementation: here, we think it is worthwhile mentioning the problem of "uninitialized local variables". Default OSR extractor does not consider, in the JVM scope descriptor, those variables that are not active at the specified bytecode index. Nevertheless, these local variables have their space allocated in the stack and this fact should be taken into account when that frame is re-established at the destination.

To summarize, in our mobility framework threads are serialized in a strong fashion: the `MobileThread` object is serialized as a regular object, while the execution state is transferred as a chain of fully serializable `MobileFrame` objects (produced by multiple invocations of the `extractSingleFrame()` method of Fig. 9). All these frames comprise data that can be deserialized by any JVM, because they are strictly bytecode-level. In the case of *Mobile JikesRVM*, this representation has been used to allow thread migration among instances of JikesRVM built for two different architectures (i.e. PPC32 and IA32), as explained in the following paragraph.

4.2.6. Resuming a migrated thread

The symmetrical part of the migration process is the creation, at the destination host, of a local instance of the migrated thread, initialized with the migrated frames. This task is appointed to the service input-channel threads that listen for migratory threads coming from the network. The entire process has been summarized earlier, but here we are going to see how the thread is rebuilt in JikesRVM.

```
void installStack() {
   // omitted auxiliary local variables

   /* 1. Compute the required space for the new stack to allocate */
   for(int i=0;i<frameSet.size();i++){
       frame = (MobileFrame) frameSet.get(i);
      userFrameSpace+=computeRealSize(frame);
   }

   stackSpace = fixedFramesSizes[0] /*preamble*/ + userFrameSpace +
   fixedFramesSizes[1] /*thread-switch part*/;

   /*2. With the computed space, allocate a new stack */
   byte[] newStack = MM_Interface.newStack(stackSpace,false);

   /*3. Attach the top frames for suspension */
   attachTopFrames(newStack);

   /*4. Install every MobileFrame read from the socket stream*/
   for(int i=0;i<frameSet.size();i++) {
       frame = (MobileFrame) frameSet.get(i);
      installFrame(frame);
   }

   /*5. Copy the bottom frames to complete stack*/
   copyBottomFrames(newStack);
}
```

Fig. 11. The stack installation phases in a code excerpt.

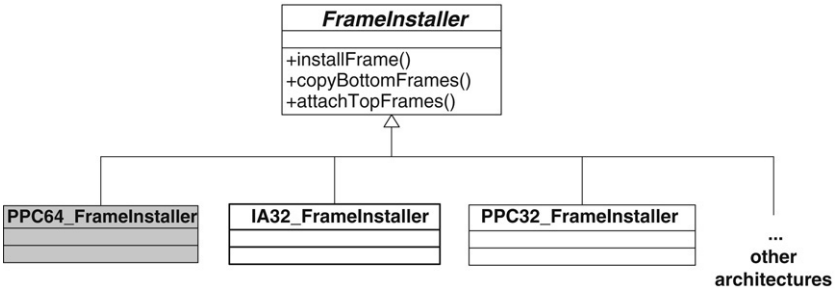

Fig. 12. The extensible design of `FrameInstaller`.

The first operation consists of creating a thread whose only task is to start execution and auto-suspend. This allows the infrastructure to safely reshape the current stack object of this thread, injecting one by one all the frames, belonging to the arrived thread. In more details, a new stack is allocated and it is filled with the thread-switch internal frames, taken from the auto-suspended thread. Then, every `MobileFrame` object is installed, in the same order as they were read from the socket stream (i.e. from the `Methodn()` to `run()`, looking at Fig. 8), to re-establish the right user-pushed frames. The brand-new stack is closed with the remaining preamble frames, again borrowed from the auto-suspended thread. The code in Fig. 11, taken from the `FrameInstaller` component, shows the above phases. Now, the new stack has been prepared and the context registers are properly adjusted (pointers are updated to refer to the new stack memory). This stack takes the place of the old stack belonging to the auto-suspended thread (the old one is discarded and becomes "garbage"). The new `MobileThread` object, with its execution state completely re-established, can be transparently resumed and continues from the next instruction.

4.2.7. Inter-platform thread migration

The mobility layer has an integrated support for multiple architectures, thus being capable, for instance, of executing a Java thread, which was born on a IA32 machine, on a PPC32 machine (and vice versa). This is possible because the above mentioned `FrameInstaller` component can adapt its behaviour according to the destination architecture. Such component has been designed as a general-purpose installer with some abstract methods (e.g. the `installFrame()` called in Fig. 11), and it can be easily extended with platform-specific stack installation functionalities.

In Fig. 12, we have reported the UML diagram of `FrameInstaller` and some of its concrete subclasses, each implementing the inherited abstract methods according to the underlying platform. The frame installation phase is the only platform-dependent one, because the memory layout of the stack and the structure of frames are tightly bound

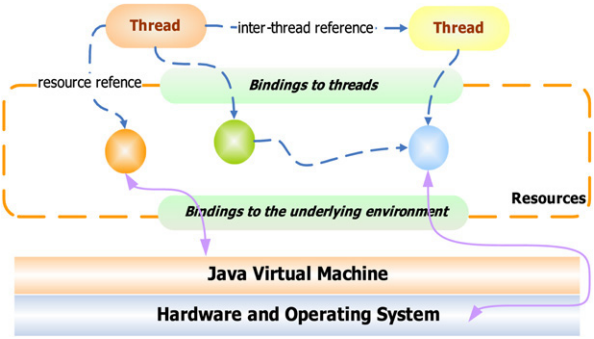

Fig. 13. A conceptual view of resources and threads.

to the kind of processor (i.e. RISC, CISC), to the number of hardware registers (e.g. used to pass parameters among methods), to the calling conventions, etc. Currently JikesRVM can run on Linux hosts with IA32 processors, but also on AIX and MacOS X with PPC32 and PPC64 processors. We have successfully produced frame installers for IA32 and PPC32, but other extensions can be added in the future (when maybe JikesRVM will support new hardware and software platforms, like Microsoft Windows OS). Nonetheless, diving into details of each specific implementation would be outside the scope of this paper. Summing up, all the platform dependent code has been concentrated into the `FrameInstaller` class, in order to grant as much portability as possible to the framework. This was possible because, for instance, the OSR is available both on Intel and PowerPC and our work consisted mainly in converting "method activation frames" into bytecode-level `MobileFrame` objects to be serialized elsewhere outside the original JVM.

4.3. The resource management layer

The set of all referenced objects of a thread has been previously defined as its *data space* [12] and, at any point during the execution, is composed of all the objects that can be reached by the thread through the call stack or its fields. As concerns the stack, the space that the thread is supposed to bring with itself comprises all the objects pointed by the parameters and local variables of methods, together with those objects pushed on the operand stack of each frame in the stack.

Although issues, like resource relocation and binding reconfiguration, pertain more to the application than to the thread migration middleware, we claim that their importance demands some kind of tool or support from a framework layer, in order to present a coherent set of mobile computing abstractions. In this section, we sketch our ideas of *MobileResources* and relocation policies. Our conceptual view of resources is depicted in Fig. 13: Java threads can have references to either active (i.e. other threads) or passive resources (i.e. regular Java objects in the heap). The bindings to the needed resources must be properly rearranged to maintain accessibility and consistency when the computation migrates to new locations. This poses two kinds of problems:

1. *handling the bindings of resources to their underlying execution environment*. This is not a problem if we consider only resources that are not bound to any OS physical entity, like pure Java objects; on the contrary, resources, such as files, sockets or database objects, cannot be serialized as they are, without carefully managing their binding to the underlying environment.
2. *handling the binding of resources to migratory threads*. Fuggetta et al. [12] identified three typologies for this bindings (*by identifier*, *by value* or *by type*) and proposed some relocation strategies for each of them (*by move*, *by copy*, *by network reference*).

As for the first point, it must be pointed out that moving some resources (e.g. a centralized database) may be neither technically (e.g. the bandwidth is not enough for its size) nor semantically (e.g. it is already in use for queries by other threads) possible. We think that such issues should be coped with by explicitly introducing the *MobileResource* concept in our programming model and letting the programmer specify the right policy for her resources. Introducing the `MobileResource` entity as an interface, the programmer will be asked to make its resource objects implement such interface, together with a set of useful methods for:

1. extracting the resource from its environment in a portable/serializable format (if the resource is fixed an exception will be raised and caught by the framework);
2. attaching the resource to the destination environment;
3. performing a correct cleanup of the resource, if it is detached from the source JVM (see the proposal by Park and Rice [20]).

A simple example of a resource can be a `java.io.File` object. A mere serialization of such an object will not produce the actual movement of the underlying file system object. To accomplish this task, the programmer has to introduce its `MovableFile` object, inheriting from `File` and implementing the `MobileResource` interface, with some of the methods detailed above: in particular, calling the "extraction method" will likely return a `byte[]` filled in with the file content; calling the "attach method" will recreate that file in the file system at destination, with its previous content.

Focusing on the second point above, the problem of the *bindings between resources and migratory threads* should be addressed. The choice of the right re-binding strategy depends on several factors, from run-time conditions and access-device properties to management requirements and user properties. For instance, a fixed server with no strict constraints on network bandwidth or memory could copy or move the needed resources and work on them locally, whereas a wireless-enabled laptop might want to access that resource remotely without moving it. However, the programming language adopted usually determines the binding strategy. Moreover, the strategy is typically embedded within the mobile application code, thus limiting *binding-management flexibility*. The resource management layer gives the programmers the means to specify which reference management policy to use, on a per-instance basis.

Relocation policies are strategies to adopt when the thread migrates and the framework has to reconfigure all the bindings to its referred objects. Three relocation strategies are allowed:

1. *by copy*: regular objects (i.e. those not implementing the `MobileResource` interface) are by default relocated "by copy", i.e. they are serialized into the object stream and therefore they have to be Java serializable objects. `MobileResources` must instead implement the inherited `extractState()` method, called by the framework to obtain the serializable state of the object to send (e.g. the file content of previous example).
2. *by move*: the object is extracted and serialized as in the "by copy" strategy. Nevertheless, the framework calls the inherited `MobileResource.detach()` method on the resource instance, to let the object carry out clean-up operations [20], such as closing handles on other resources (e.g. calling the `close()` method on a `File` or a `Socket` object).
3. *by ref* (i.e. by network reference): the real object is not serialized, but it remains attached to the source environment. A proxy object (instance of `MobileResourceRef`) is instead serialized and it is used at destination exactly as the real object. Field accesses and method calls on such object are transparently forwarded by the framework to the real object on the previous host. More details on this policy are provided in the next paragraph.

As shown in Fig. 14, each thread can register its policies on its resources (by means of `MobileThread.setPolicy()` method) and these policies are local to the registering thread. These registration can be modified (a new registration on the same object overwrites the old one) throughout the life of the thread, depending on the needs of the application. When the thread decides to move, the mobility layer (see Fig. 3) asks the resource management layer to apply the registered policy on each referenced resource and it obtains the serializable representation for that resource. Once at destination, the thread is deserialized and every resource is reconnected according to the chosen policy.

It should be noted that the resource management layer is able to properly deal with three non-negligible kinds of issues:

1. *conflicts among registered policies*: if one thread calls the `setPolicy()` method to register its relocation policy on object A, then the system verifies the compatibility of the new policy with the others registered by other threads on A. If the verification fails, an exception is raised by `setPolicy()` to the registering thread. In our experience with the above relocation policies, the only incompatible couple is ("by ref", "by move"), as in Fig. 14.
2. *inter-thread references*: references to non-mobile threads are treated as references to regular objects, except for their incompatibility with both the "by copy" and "by move" relocation policies (because they are not serializable at all). As for mobile threads, the latter two policies are allowed: when a reference to `MobileThread` T2 is found, the framework attempts to reactively migrate it and transparently reconfigures the binding at destination. If migration fails, an exception is notified to the first thread and the migration process is aborted.

```
public class MyThread1 extends MobileThread {
   private MyResource r;
   private MyThread2 t;
   public void run(){
      r = new MyResource();
      t.setResource(r);
      try{ setPolicy(r,BY_REF);
      // …
      migrate(host,port);
      // resumed --> uses the resource remotely
      r.field1="Hello"; } catch(…){…}
      // …
   }
}

public class MyThread2 extends MobileThread {
   public int setResource(MyResource r){
      MyResource r1 = r;
      try {setPolicy(r1, BY_MOVE);} catch(..){…}
      // perform some operation with the resource
   }
}
```

Fig. 14. Two threads registering policies on the same object.

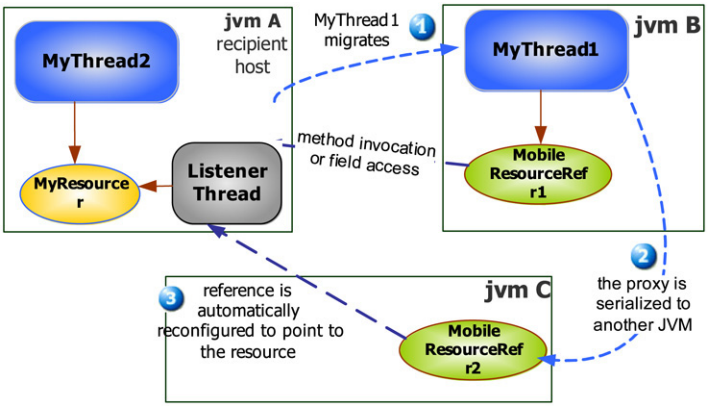

Fig. 15. Resources and remote proxies (example of Fig. 14).

3. *synchronization upon resources*: locks acquired upon objects are automatically handled by the resource management layer, according to the following considerations. Firstly, in presence of a regular object (not a resource) or a resource registered "by copy", the migratory thread transparently releases its lock on the real object and reacquires it on the copy at destination. If "by move" is chosen, the behaviour is the same as "by copy", if no other thread is currently synchronized on that object. Otherwise, an exception abruptly terminates the migration, as it would be quite an unfair strategy to move an object upon which other threads are currently synchronized. A "by ref" lock is remotely managed as detailed in the following paragraph.

4.3.1. More on the "By Ref" relocation policy

Implementing the "by ref" policy requires hiding the details of the physical distribution as much as possible, in order to guarantee maximum *transparency* to the programmer: the thread can access fields and methods of the remote object, just like it would do with a local instance.

Java RMI provides only a partial support to our purpose, mainly because it allows clients to only invoke methods on remote objects, while fields cannot be directly accessed (the remote object should expose proper `setXX()` and `getXX()` methods to achieve this). Moreover, RMI does not provide the degree of transparency we are seeking, because it requires proxies and stubs to be built and compiled when a new kind of remote object is accessed. Thus, we chose to adopt a more purpose-built protocol than RMI, excluding features such as the "naming service" and the registry, and adding the "field access" feature and others.

The resource management layer is capable of handling remote references thanks to a proper `MobileResourceRef` object (see Fig. 15), which is just a proxy to the remote object. It has exactly the same fields and methods of its remote

counterpart. Furthermore, the resource management layer keeps track of the hashcode of the remote object and the hostname where it is located (we will call it "recipient host"), so that:

1. every access (through Java `getfield` and `putfield` bytecodes) to its fields triggers a communication on a socket with the recipient host, where a service thread listens to remote access requests on a predefined port.
2. every invoked method is implemented as a remote invocation (not an RMI invocation), served by the previous service thread.

One important difference is that the programmer should be aware of the fragility of such a reference, protecting both field accesses and method invocations with try/catch blocks (to deal with network failures). If a `MobileResourceRef` is passed as a parameter of a remote method, is set as a field value or as method return value (see step 2 of Fig. 15), the framework is able to properly handle the situation and creates another proxy on the destination (e.g. r2 on JVM B), which points to the recipient host and not to the original proxy (e.g. r1); if the proxy is passed back to the initial JVM (e.g. JVM A), the framework converts the proxy back to a reference to the real object. This behaviour avoids problems with circular remote references among proxies.

Synchronization on remote objects is also transparently handled, acquiring and releasing locks thanks to a dedicated service thread, running on the recipient host. Thus, entering a synchronized block on a `MobileResourceRef` object, means that a service thread on the recipient host will try to acquire the lock on the referenced object and possibly wait for it. The resource management layer automatically releases the lock when the socket connection with the recipient host is closed for some reason (to avoid indefinite blocking). As concerns the garbage collectors, it is guaranteed by the resource management layer that, when the thread has no more references to the `MobileResourceRef` proxy, the recipient host will be notified and the remote reference broken.

4.3.2. By ref implementation on JikesRVM

The resource management layer has been successfully realized in JikesRVM, thanks to its extremely extensible architecture and, in particular, its JIT compiler, which can be easily patched by researchers (as we said for yieldpoints), who want to experiment new Java techniques on a widely used JVM. The extension implemented in our resource management layer practically introduces the `MobileResourceRef` concept: whenever a `MobileResource` is relocated "by ref", its binding to the migrated thread is transparently reconfigured, so that every access to fields or methods of such resource is forwarded (through the network) to the recipient host.

As for field access, we have extended the behaviour of the `getfield` and `putfield` Java language bytecodes, so that JikesRVM JIT compiler can properly handle the case of a remote resource. For instance, let us consider the `getfield` bytecode, which should retrieve the value of a field, leaving it on top of the operand stack [15]. Our extended JIT compiler is able to recognize `MobileResource` references, inserting a hidden method call in the compiled code whose functionality can be summarized as follows:

1. query the resource management layer to know if the current `MobileResource` object is just a proxy or a real resource object;
2. if it is a remote proxy, then send a "getfield" request to the listener thread at the destination (see Fig. 15) and wait until the value is ready;
3. save the retrieved value in the field of the proxy object.

Without knowing what happens beneath the surface, the migrated thread can get any fields of a resource relocated *by ref*, just like it would have done with a local object. There are obviously some slight differences that no layer can hide and they pertain to the usage of the network: one observation is that it is inevitable that accessing a remote field is slower than doing the same thing locally, because it implies paying the price of distribution; another crucial aspect is related to network failures which are responsibility of the programmer using a "by ref" relocated resource. Similar considerations are applicable to the `putfield` bytecode, whose description is omitted due to space limitations.

Method invocations are instead forwarded at the recipient host, simply manipulating the proxy *TIB* (*Type Information Block*). The TIB in JikesRVM [1] is a sort of "method dispatch table" and it is referred to by an header within each instantiated object. The resource management layer modifies the TIB of the proxy object, when it is created on the destination JVM, forcing it to point to a special "remote invoker" method. The latter method simply posts an invocation request to listener thread on the recipient host and waits for the results to come back.

Finally, the migrated thread can synchronize on a remote resource, thanks to the special handling of `synchronized` methods and of `monitorenter` and `monitorexit` bytecodes, as described above in this section.

Table 1
Evaluated times for thread serialization (sec.)

	5 frames	15 frames	25 frames
OSR Frame capturing	1.78E−5	1.89E−5	1.96E−5
State building	3.44E−5	3.75E−5	3.43E−5
Pure serialization	2.49E−3	7.32E−3	1.50E−2
Overall times	2.54E−3	7.38E−3	1.51E−2

Table 2
Evaluated times for thread rebuilding (sec.)

	5 frames	15 frames	25 frames
Pure de-serialization	4.46E−3	5.33E−3	7.06E−3
State rebuilding	5.45E−4	5.27E−4	5.06E−4
Stack installation	1.53E−3	1.60E−3	1.71E−3
Overall times	6.54E−3	7.46E−3	9.28E−3

5. Performance and evaluation

At the current stage of our project (whose code is available at [18]), the thread serialization mechanism, discussed so far, has been successfully tested, focusing mainly on the times needed for state capturing and restoring. We made, therefore, some performance tests to discover possible bottlenecks and evaluate the cost of each migration phase. We wrote a benchmark based on the Fibonacci recursive algorithm, just to see the variation of migration times with respect to an increasing number of frames on the stack: consider that this time can vary depending on the kind of method to be captured and that is the main reason why we have chosen a recursive algorithm for our tests (although we are currently trying more complete benchmarks). The times measured are expressed in seconds and are average values computed across multiple runs, on a Pentium IV 3.4 GHz, 1 GB RAM, JikesRVM release 2.4.6 (built using the "prototype-opt" configuration). Some sample times (taken with a number of 5, 15 and 25 frames) are listed in Tables 1 and 2 and they demonstrate very graceful time degradation. The times have been conceptually divided into two phases, where Table 1 refers to the thread serialization phase, while Table 2 refers to the corresponding de-serialization phase at the destination host (the transfer time through the network has not been considered significant here and so it is not reported).

Considering how these times are partitioned among the different phases of the thread serialization, we can see that the bulk of the time is wasted in the pure Java serialization of the captured state (which is completely due to the Java serialization implementation), while the frame extraction mechanism (i.e. the core of our entire facility) has very short times instead.

The same bottleneck due the Java serialization may be observed in the de-serialization of the thread.

In the latter case, however, we have an additional foreseeable time in the stack installation phase, since the system has often to create a new thread and compile the methods for the injected frames. These performance bottlenecks can be further minimized, perhaps using externalization to speed up the serialization of the thread state [29]. Java serialization uses, in fact, reflection to inspect the content of objects and serialize their field into a stream; Java reflection has the drawback of being quite slow even in commercial JVMs.

As concerns reactive migration, a JIT compiler extension was described that allows us to perform reactive migration, in addition to the simpler proactive case. We said that, in that cases, migration points are installed instead of traditional yieldpoints in some chosen method bodies. This has two negligible costs:

1. *on thread execution time*, since migration points are taken only if a thread switch is requested (they are in fact a subset of yieldpoints). However, such an approach does not suffer from the slowdown of many application-level approaches (described within the related work section): many of those systems inject checkpoint instructions into the bytecode, to trace the execution state and let the thread migrate only in predefined points. We do not, instead, insert any additional checkpoint in the code, but simply extend the functionality of normal pre-existent JikesRVM yieldpoints.

2. *on method JIT compilation time*, since the additional test to be performed on the candidate method at JIT compilation time is very simple. Referring again to Section 4.2, the extended `genThreadSwitchTest()` method performs three simple necessary tests, to determine if the method needs the insertion of our migration points or the old JikesRVM yieldpoints.

No other overheads are imposed on JikesRVM normal performances.

6. Future work

Additional features can be, of course, implemented to extend our thread mobility framework in the future. First of all, the optimized compiler is not fully supported yet. OSR can extract the JVM scope descriptor even from optimized method frames, but this requires some cooperation from the optimizing compiler to generate mapping information needed to correctly interpret the structure of the optimized frame: in fact, while baseline frames have a fully predictable layout, the same is not true for optimized ones where local variables can be allocated into machine registers, pieces of code can be suppressed or inlined, and so forth. For these reasons, the optimizing compiler choices some points in the code where OSR can occur and, just for these points, maintains all the necessary mapping information. Such points, called *OSR Points*, do not include "method call sites" and for that reason our serialization system cannot capture optimized frames yet. Nevertheless, we are working at another compiler patch, capable of inserting OSR Points even at "method call sites". A similar effort has been done in another context [24], where Krintz and Soman tried to present a more general-purpose version of OSR that is more amenable to optimizations than the current one. This OSR improvement will thus be exploited to perform a truly complete thread state capturing, even in presence of code optimizations.

Future work includes also a comparison with other proposed thread migration systems, to improve our performance evaluation understanding and identify possible undetected bottlenecks.

7. Conclusions

This paper has presented our *Mobile JikesRVM* framework that extends the facilities provided by the IBM JikesRVM in order to support Java thread strong migration. Thanks to its modular design and its minimally intrusive nature, the developed framework can be easily adopted in distributed application developments, provided that an OSR-enabled version of JikesRVM is installed in the system (it can be classified as a midway approach between the described JVM-level and the application-level approaches). Users do not have to download a modified, untrustworthy, version of JikesRVM, but can import the implemented mobility package into their code and execute it on their own copy of JikesRVM. Moreover, thanks to the support for resource relocation strategies, inter-threads reference management and cross-platform implementation, *Mobile JikesRVM* provides the programmer with a full-fledged set of tools to address code mobility in her complex distributed applications (source code for JikesRVM 2.4.6 is freely available visiting the project web site [18]).

Acknowledgement

This work was supported by the European Community within the EU FET project "CASCADAS".

References

[1] B. Alpern, D. Attanasio, J.J. Barton, A. Cocchi, S.F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, S. Smith, Implementing Jalapeño in Java, in: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '99, Denver, Colorado, November 1, 1999.

[2] B. Alpern, C.R. Attanasio, D. Grove, et al., The Jalapeno virtual machine, IBM System Journal 39 (1) (2000).

[3] M. Arnold, S. Fink, D. Grove, M. Hind, P.F. Sweeney, Adaptive optimization in the Jalapeño JVM, in: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2000, Minneapolis, Minnesota, October 15–19, 2000.

[4] A. Balachandran, G.M. Voelker, P. Bahl, Wireless hotspots: Current challenges and future directions, in: Mobile Networks and Applications Journal, Springer Science Publishers, 2005, pp. 265–274.

[5] S. Bouchenak, D. Hagimont, N. De Palma, Efficient Java thread serialization, in: 2nd ACM International Conference on Principles and Practice of Programming in Java, Kilkenny, Ireland, June 2003.

[6] S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma, F. Boyer, Experiences implementing efficient Java thread serialization, Mobility and Persistence, I.N.R.I.A., Research report no. 4662, December 2002.
[7] S. Bouchenak, D. Hagimont, S. Krakowiak, N. De Palma, F. Boyer, Experiences implementing efficient java thread serialization, mobility and persistence, Software — Practice & Experience 34 (4) (2004).
[8] G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, The Jalapeno dynamic optimizing compiler for Java, in: ACM Java Grande Conference, June 1999.
[9] C. Chambers, The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages, Ph.D. Thesis, Stanford University, March 1992. Published as tech. report STAN-CS-92-1420.
[10] G. Cabri, L. Leonardi, F. Zambonelli, Weak and strong mobility in mobile agent applications, in: Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java, PA JAVA 2000, Manchester (UK), April 2000.
[11] S. Fink, F. Qian, Design, implementation and evaluation of adaptive recompilation with on-stack replacement, in: International Symposium on Code Generation and Optimization San Francisco, CA, March 2003.
[12] A. Fuggetta, G.P. Picco, G. Vigna, Understanding code mobility, IEEE Transactions on Software Engineering 24 (1998).
[13] S. Funfrocken, Transparent migration of Java-based mobile agents (capturing and reestablishing the state of Java programs), in: 2nd International Workshop on Mobile Agents 98, MA'98, Stuttgart, Germany, September 1998.
[14] http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html.
[15] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, second edition, SUN Microsystem.
[16] T. Illmann, T. Krueger, F. Kargl, M. Weber, Transparent migration of mobile agents using the Java platform debugger architecture, in: Proceedings of the 5th International Conference on Mobile Agents, Atlanta, GA, USA, December 2001.
[17] M. Kim, D. Kotz, S. Kim, Extracting a mobility model from real user traces, in: Proceedings of the 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), Barcelona, Spain, April, 2006.
[18] Project's web site at: http://www.agentgroup.unimore.it/Quitadamo.
[19] The 19 Project. http://openmosix.sourceforge.net/.
[20] D. Parka, S. Rice, A framework for unified resource management in Java, in: The Proceedings of the International Conference on Principles and Practices of Programming In Java, PPPJ 2006, Mannheim, Germany, August 30–September 1, 2006.
[21] R. Quitadamo, G. Cabri, L. Leonardi, Strong agent mobility for aglets based on the IBM JikesRVM, in: The Proceedings of the ACM Symposium on Applied Computing, SAC'06, Dijon, France, April 2006.
[22] T. Sakamoto, T. Sekiguchi, A. Yonezawa, A bytecode transformation for portable thread migration in Java, in: 4th International Symposium on Mobile Agents 2000, MA'2000, Zurich, September 2000.
[23] T. Sekiguchi, A. Yonezawa, H. Masuhara, A simple extension of java language for controllable transparent migration and its portable implementation, in: 3rd International Conference on Coordination Models and Languages, Amsterdam, The Netherlands, April 1999.
[24] S. Soman, C. Krintz, Efficient and general on-stack replacement for aggressive program specialization, in: International Conference on Programming Languages and Compilers (PLC), Las Vegas, NV, June 2006.
[25] T. Suezawa, Persistent execution state of a java virtual machine, in: ACM Java Grande 2000 Conference, San Francisco, CA, USA, June 2000.
[26] A. Acharya, M. Ranganathan, J. Saltz, Sumatra: A language for resource-aware mobile programs, in: 2nd International Workshop on Mobile Object Systems, MOS'96, Linz, Austria, 1996.
[27] N. Suri, et al. An overview of the NOMADS mobile agent system, in: Workshop On Mobile Object Systems in Association with the 14th European Conference on Object-Oriented Programming, ECOOP 2000, Cannes, France, 2000.
[28] A.S. Tanenbaum, M. Van Steen, Distributed Systems: Principles and Paradigms, Prentice Hall PTR, NJ, USA, 2001.
[29] Sun Microsystems. Improving Serialization Performance with Externalizable. http://java.sun.com/developer/TechTips/txtarchive/2000/Apr00_StuH.txt.
[30] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, P. Verbaeten, Portable support for transparent thread migration in Java, in: 4th International Symposium on Mobile Agents 2000, MA'2000, Zurich, Switzerland, September 2000.
[31] X. Wang, Translation from strong mobility to weak mobility for Java, Master's Thesis, The Ohio State University, 2001.
[32] W. Zhu, C. Wang, F.C.M. Lau, JESSICA2: A distributed Java virtual machine with transparent thread migration support, in: IEEE Fourth International Conference on Cluster Computing, Chicago, USA, September.